Reducing the Memory Requirement in Reverse Mode Automatic Differentiation by Solving TBR Flow Equations

Uwe Naumann

Mathematic and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA

Abstract. The fast computation of gradients in reverse mode Automatic Differentiation (AD) requires the generation of adjoint versions of every statement in the original code. Due to the resulting reversal of the control flow certain intermediate values have to be made available in reverse order to compute the local partial derivatives. This can be achieved by storing these values or by recomputing them when they become required. In any case one is interested in minimizing the size of this set.
Following an extensive introduction of the "To-Be-Recorded" (TBR) problem we will present flow equations for propagating the TBR status of variables in the context of reverse mode AD of structured programs.

1 Introduction

The work presented here is a continuation of the results published in [5]. Our aim is to motivate a more formalized view on the problem of generating adjoint code using the reverse mode of AD [9] that requires a minimal amount of memory space when following a "store all" *taping* strategy, which will be explained below.

We consider a single subroutine $F : \mathbb{R}^n \to \mathbb{R}^m$ for computing a vector function $\mathbf{y} = F(\mathbf{x})$. The values of m dependent variables y_j, $j = 1, \ldots, m$, are calculated from the n independent variables x_i, $i = 1, \ldots, n$. F represents an implementation of the mathematical model for some underlying real-world application and it will be referred to as the forward code. The forward code is expected to be written in some high-level imperative programming language such as C or Fortran. More generally, it should be possible to decompose F into a sequence of scalar assignments of the form

$$v_j = \varphi_j(v_k)_{k \prec j}, \quad j = 1, \ldots, p + m, \tag{1}$$

such that the result of every intrinsic function and elementary arithmetic operation is assigned to a unique intermediate variable v_j, $j = 1, \ldots, p + m$. m out of these intermediate variables are set to be dependent. Whenever some variable v_j depends directly on another variable v_k we write $k \prec j$. It is assumed that the local partial derivatives

$$c_{ji} = \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec i} \tag{2}$$

of the *elemental* functions φ_j, $j = 1, \ldots, p+m$, exist and that they are jointly continuous in some open neighborhood of the current argument $(v_k)_{k \prec i}$. In this case an augmented version of the forward code can be implemented that computes F itself and the set of all local partial derivatives as defined in (2).

The reverse mode of AD (see for example Section 3.3 in [9]) uses these local partial derivatives to compute adjoints

$$\bar{v}_k = \sum_{j:k \prec j} c_{jk} \cdot \bar{v}_j, \quad j = p, \ldots, 1-n. \tag{3}$$

"Transposed Jacobian matrix times vector" products $\bar{\mathbf{x}} = F'(\mathbf{x})^T \cdot \bar{\mathbf{y}}$ are computed by initializing the adjoints of the dependent variables $\bar{y}_j \equiv \bar{v}_{p+j}$, $j = 1, \ldots, m$. The Jacobian $F'(\mathbf{x})$ can be accumulated by reverse propagation of the Cartesian basis vectors in $\mathbb{R}^m$ at complexity $O(m)$. In particular, gradients of single dependent variables with respect to all independent variables can be obtained at a computational cost that is a small multiple of the cost of running the forward code (see *cheap gradient principle* in [9]).

Considering equation (3) we observe that in reverse mode AD the adjoints of all intermediate variables are actually computed in reverse order, i.e. for $j = p, \ldots, 1-n$. This implies that the local partial derivatives c_{jk} have to be made available in reverse order as well. This can be ensured

1. by storing the arguments of all local partial derivatives on a so-called *tape* before their values get overwritten during the execution of the augmented forward code and by retrieving these values whenever required in the adjoint code [5]
2. or by simply recomputing them "from scratch" [7] when they become required in the adjoint code.

Generally speaking, the arguments of all local partial derivatives have to be *recorded*. We will use this term as a place holder for either 1. or 2. Obviously, the former approach may lead to enormous memory requirements for large-scale application programs whereas the latter results in a quadratic computational complexity. Often a mixture of both strategies is employed to achieve reasonable trade-offs between memory requirements and the number of floating-point operations. However, even the efficiency of these *checkpointing* schemes [8] depends on the knowledge about whether some value is actually required or not.

2 TBR Problem

The generation of the adjoint model is done by associating adjoint components $\bar{v}$ with every *active* variable v. In particular, both the independent variables $\mathbf{x}$ and the dependent variables $\mathbf{y}$ are active. Here, the term variable should be understood as a scalar component of some *program variable* that is actually declared in the forward code. An intermediate variable v is active at a given point within the program if $\exists x \in \mathbf{x} : x \prec^* v$ and $\exists y \in \mathbf{y} : v \prec^* y$. Here, $\prec^*$ denotes the

transitive closure of the operator $\prec$, i.e. $x \prec^* v$ if there exist $v_0, v_1, \ldots, v_{k-1}, v_k$ such that $x = v_0 \prec v_1 \prec \ldots \prec v_{k-1} \prec v_k = v$. In the following, we assume that the information on the set of active variables is available at every single point in the program. A variable that is not active is called *passive*.

We investigate reverse mode AD of structured programs [3, Section 10] by concentrating on the following four different types of statements:

- $s := [v = f(\mathbf{u})]$ – scalar assignments with $f : I\!R^k \to I\!R$ as they occur in most imperative programming languages; this restriction helps to keep the notation simple; all results can be generalized for general (vector) assignments $f : I\!R^{k_1} \to I\!R^{k_2}$ as they exist for example in Fortran 95 [1];
- $s := [s_1,\ s_2]$ – cascades of statements;
- $s := [\texttt{if}\ (c)\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}]$ – branches where the boolean value c determines whether s_1 or s_2 is executed;
- $s := [\texttt{while}\ (c)\ \texttt{do}\ s_1\ \texttt{done}]$ – loops where c determines if s_1 is executed followed by another evaluation of c.

$\mathbf{u}$ will be considered as a set of scalar variables, i.e. we will write $w \in \mathbf{u}$ whenever the scalar variable w occurs on the right-hand-side of an assignment $v = f(\mathbf{u})$. c is the value of a scalar boolean function $g : D \to \{\texttt{true}, \texttt{false}\}$ over elements of arbitrary data types, i.e. $c = g(\mathbf{v})$ and the values of the arguments of the g determine the value of c and therefore the control flow.

In order to generate a correct adjoint code one has to do the following:

1. The control flow of the forward code has to be reversed.
2. Adjoint versions of every single assignment have to be built.

The former can be achieved in various ways. An exhaustive discussion of these issues is out of the scope of this paper. In the example presented in Section 3 we have chosen the following approach:

- Loops $s := [\texttt{while}\ (c)\ \texttt{do}\ s_1\ \texttt{done}]$ are reversed by counting the number `ITER` of iterations performed when running the forward code and by executing the adjoint of the loop body s_1 exactly `ITER` times.
- For branches $s := [\texttt{if}\ (c)\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}]$ we push the values of all arguments of c onto the tape whenever they get overwritten during the execution of the forward code. When running the adjoint code these values are popped at the appropriate time to decide whether to execute the adjoint version of s_1 or s_2. If such an argument is overwritten inside s_1 or s_2 then its value has to be retrieved before the execution of of the adjoint branch. This is automatically the case if the overwriting takes place after the execution of s in the forward code.

We do not claim this solution to be optimal. However, for structured programs it is a simple method for ensuring a correct reversal of the control flow.

In this paper we will concentrate on the second crucial ingredient of an adjoint code, namely the generation of adjoint versions for all assignments in the forward code. Consider $v = f(\mathbf{u})$ where $\mathbf{u} = \{u_1, u_2, \ldots, u_{n_f}\}$ denotes the set of scalar

arguments of f. Reverse mode AD transforms this assignment into the set of adjoint statements

$$\bar{u}_i = \bar{v} \cdot \frac{\partial f}{\partial u_i}(\mathbf{u}), \quad i = 1, \ldots, n_f.$$

Three types of values are required for evaluating them correctly:

1. $\mathrm{args}(f'(\mathbf{u}))$ – the arguments of the local partial derivatives $\frac{\partial f}{\partial u_i}(\mathbf{u})$ for $i = 1, \ldots, n_f$;
2. $\mathrm{idxargs}(\mathbf{u})$ – arguments of indices of array-type u_i, $i \in \{1, \ldots, n_f\}$;
3. $\mathrm{idxargs}(v)$ – arguments of indices of v should v be an array element.

A more detailed characterization of $\mathrm{args}(f'(\mathbf{u}))$ has been given in [5] as follows: The TBR status of $w \in \mathbf{u}$ has to be activated if

1. w is a non-linear active argument, e.g. $v = \sin(w)$;
2. w is a passive argument in an active term, e.g. $v = w \cdot a$ where a is an active variable;
3. w is the index of some active element of an array a which occurs non-linearly on the right-hand-side, e.g. $v = a(w+1) \cdot a(w)$;
4. w is the index of some passive element of an array p which occurs in an active term, e.g. $v = z(w) \cdot a$ where a is some active variable;

For a better understanding of this rule it is useful to notice the following comments:

- The decision whether some individual element of an array is active is impossible to make in general. Both static and dynamic *array region analysis* [10] can help to compute some conservative estimate. Without it the activity of one element implies the activity of the whole array.
- 3. actually describes a subset of $\mathrm{idxargs}(\mathbf{u})$. The correct value of the array index is required for the adjoint as well as for restoring the original value that enters the computation of the local partial derivative.
- The indices of passive array elements are only required for restoring the correct arguments of the local partial derivatives. No adjoints are associated with passive variables.

For $s := [v = f(\mathbf{u})]$ we define

$$\mathbf{TBR}(s) = \mathrm{args}(f'(\mathbf{u})) \cup \mathrm{idxargs}(\mathbf{u}) \cup \mathrm{idxargs}(v).$$

Knowing how to compute $\mathbf{TBR}(s)$ for all assignments of F we are able to decide whether the value of some variable has to be recorded. Such variables will be referred to as "tbr-active".

3 Example

Consider the following code fragment (original forward code in lower case on the left-hand-side) which has been augmented by instructions for storing the tape

on the right (new statements in upper case). It can be wrapped into a subroutine F computing new values for the elements of a vector `x` from the corresponding input values.

forward code	**augmented forward code**

`i=0; j=10`
 	
`while (check(j)) do`
 	
`  if (max(i,j)>7) then`
 	
`    x(i)=j+sin(x(i))`
`  else`
 	
`    x(j)=j*cos(x(j))`
`  fi`
 	
`  i=i+1`
 	
`  j=j-1`
`done`
	`i=0; j=10`
	`ITERS=0`
	`while (check(j)) do`
	`  ITERS=ITERS+1`
	`  if (max(i,j)>7) then`
	`    STORE(x(i))`
	`    x(i)=j+sin(x(i))`
	`  else`
	`    STORE(x(j))`
	`    x(j)=j*cos(x(j))`
	`  fi`
	`  STORE(i)`
	`  i=i+1`
	`  STORE(j)`
	`  j=j-1`
	`done`

`i` and `j` are assumed to be integers. The control flow is determined by the two boolean values $c_1 := \texttt{check(j)}$ and $c_2 := \texttt{max(i,j)} > 7$ where `check` is some boolean function over the integers and `max` computes the maximum of two numbers. `STORE(`w`)` puts the current value of the variable w onto the top of the stack implementing the tape for variables of the same type as w.

Let us have a closer look at the augmented version of the forward code. The integer variable `ITER` is introduced to count the number of iterations performed by the `while` loop. Both the values of i and j are required to generate the adjoint version $\bar{s}$ of the `if`-statement s and therefore $\mathbf{TBR}(c_2) = \{\texttt{i}, \texttt{j}\}$. Notice, that neither `i` nor `j` is overwritten inside s_1 or s_2. Consequently, their values do not have to be restored before the execution $\bar{s}$. The fact that they are is a consequence of both `i` and `j` being overwritten immediately after the `if`-statement in the forward code.

For the assignment $s_1 := [v_1 = f_1(\mathbf{u}_1)] \equiv [\texttt{x(i)} = \texttt{j} + \texttt{sin(x(i))}]$ we observe that

$$\begin{aligned}\mathbf{TBR}(s_1) &= \mathrm{args}(f_1'(\mathbf{u}_1)) \cup \mathrm{idxargs}(\mathbf{u}_1) \cup \mathrm{idxargs}(v_1)\\ &= \{\texttt{x(i)}\} \cup \{\texttt{i}\} \cup \{\texttt{i}\} = \{\texttt{x(i)}, \texttt{i}\}.\end{aligned}$$

Similarly,

$$\begin{aligned}\mathbf{TBR}(s_2) &= \mathrm{args}(f_2'(\mathbf{u}_2)) \cup \mathrm{idxargs}(\mathbf{u}_2) \cup \mathrm{idxargs}(v_2)\\ &= \{\texttt{x(j)}, \texttt{j}\} \cup \{\texttt{j}\} \cup \{\texttt{j}\} = \{\texttt{x(j)}, \texttt{j}\}\end{aligned}$$

for $s_2 := [v_2 = f_2(\mathbf{u}_2)] \equiv [\mathtt{x(j)} = \mathtt{j} * \mathtt{sin(x(j))}]$. In both statements the TBR status of the variable written is activated on the right-hand-side which results in the corresponding `STORE` instructions preceding the statement itself. Notice, that both `i` and `j` are stored as a result of being arguments of `max(i,j)` and as array indices of `x`. Moreover, `j` is recorded as an element of $\mathrm{args}(f_2'(\mathbf{u}_2))$.

We assume *joint program reversal mode* [9, Chapter 12] meaning that the adjoint computation is performed immediately after the execution of the augmented forward code. A possible implementation of the adjoint model is given by

adjoint code

```
while (ITERS>0) do
  RESTORE(j)
  RESTORE(i)
  if (max(i,j)>7) then
    RESTORE(x(i))
    adj_x(i)=cos(x(i))*adj_x(i)
  else
    RESTORE(x(j))
    adj_x(j)=-j*sin(x(j))*adj_x(j)
  fi
  ITERS=ITERS-1
done
```

`RESTORE`(w) puts the value from the top of the stack matching the data type of w into w. The `RESTORE` statement is always executed before the adjoint version of the statement in front of which the matching `STORE` statement was performed in the augmented forward code. In certain situations it can be advantageous to store the result of an assignment instead of its arguments (see [5]).

Given values for `x` and `adj_x` the program consisting of the augmented forward code followed by the adjoint code computes the "transposed Jacobian matrix times adjoint vector" product

$$\mathtt{adj\_x} = F'(\mathtt{x})^T \cdot \mathtt{adj\_x}.$$

Alternatively, we might have recomputed the values of `i, j, x(i),` and `x(j)` which would have lead to repeated executions of the forward code within the adjoint section while not requiring a tape.

4 TBR Status Flow Equations

In analogy to the approach taken in [3, Section 10] we will consider the following sets:

- $\mathbf{In}_p(s)$ – variables having property p before the execution of a statement s;

- $\mathbf{Out}_p(s)$ – variables having property p after the execution of s;
- $\mathbf{Gen}_p(s)$ – variables gaining property p as the result of executing s;
- $\mathbf{Kill}_p(s)$ – variables loosing property p as the result of executing s;

In particular, we are interested in the case $p = tbr$ (tbr meaning "has *t*o *b*e *r*ecorded"), i.e. in $\mathbf{In}_{tbr}(s)$, $\mathbf{Out}_{tbr}(s)$, $\mathbf{Gen}_{tbr}(s)$, and $\mathbf{Kill}_{tbr}(s)$.

The decision to be made is whether the value of a variable v overwritten by some assignment $s := [v = f(\mathbf{u})]$ is required for the evaluation of the adjoint program. If so, it has to be recorded. Below we consider assignments and basic blocks, cascades of statements, branches, and loops – each of them interpreted as a single statement s – under two aspects:

1. Which variables have to be recorded before the execution of s (for assignments only)? Under the restriction to scalar assignments the question is whether the value of the variable on the left-hand-side should be recorded or not.
2. The TBR status of which variables is active after the execution of s (for all statements)?

Assignments. For scalar assignments $s := [v = f(\mathbf{u})]$ the set $\mathbf{Kill}_{tbr}(s)$ is either empty or it contains the single element v. The latter is the case if the TBR status of v is activated before the execution of s or by s itself. Thus,

$$\mathbf{Kill}_{tbr}(s) = (\mathbf{In}_{tbr}(s) \cup \mathbf{TBR}(s)) \cap v \tag{4}$$

is exactly the set of values that would have to be saved before the execution of s as part of the augmented forward code if we were following a "store all" strategy. For sets containing a single element v only we write v instead of $\{v\}$. Intuitively, we can state that a variable v belongs to $\mathbf{Gen}_{tbr}(s)$ if it is in $\mathbf{TBR}(s)$ but neither in $\mathbf{In}_{tbr}(s)$ nor in $\mathbf{Kill}_{tbr}(s)$, i.e.

$$\mathbf{Gen}_{tbr}(s) = \mathbf{TBR}(s) \setminus \mathbf{In}_{tbr}(s) \setminus \mathbf{Kill}_{tbr}(s).$$

Which can be simplified to get

$$\mathbf{Gen}_{tbr}(s) = \mathbf{TBR}(s) \setminus \mathbf{In}_{tbr}(s) \setminus v. \tag{5}$$

Sequences of set differences are evaluated from left to right. Notice, that this operation is not associative. Both the expressions for $\mathbf{Kill}_{tbr}(s)$ and for $\mathbf{Gen}_{tbr}(s)$ are required for the resolution of

$$\mathbf{Out}_{tbr}(s) = \mathbf{Gen}_{tbr}(s) \cup (\mathbf{In}_{tbr}(s) \setminus \mathbf{Kill}_{tbr}(s)) \tag{6}$$

This standard data flow equation (see for example [3]) says that a variable is tbr-active after the execution of a statement s if its TBR status became activated by s or if it was tbr-active before s and was not made tbr-passive by s. Substituting (4) and (5) in equation (6) results in

$$\mathbf{Out}_{tbr}(s) = (\mathbf{TBR}(s) \cup \mathbf{In}_{tbr}(s)) \setminus v. \tag{7}$$

Under the restrictions imposed by us assignments s are the only statements for which we are actually interested in $\mathbf{Kill}_{tbr}(s)$. For the remaining types of statements we need to be able to compute $\mathbf{Out}_{tbr}(s)$ from $\mathbf{In}_{tbr}(s)$. Both $\mathbf{Gen}_{tbr}(s)$ and $\mathbf{Kill}_{tbr}(s)$ can be computed recursively from the underlying assignments.

Equation (7) can be generalized to become

$$\mathbf{Out}_{tbr}(s) = \bigcup_{i=1,\ldots,l} \left(\mathbf{TBR}(\mathbf{u}_i) \setminus \left(\bigcup_{j=i,\ldots,l} v_j \right) \right) \cup \mathbf{In}_{tbr}(s) \setminus \left(\bigcup_{i=1,\ldots,l} v_i \right) \quad (8)$$

for cascades of l assignments, i.e. for *basic blocks* $s := [s_i, i = 1, \ldots, l]$ where $s_i := [v_i = f_i(\mathbf{u}_i)]$ and $i = 1, \ldots, l$. Assuming that $\mathbf{In}_{tbr}(s)$ is known equation (8) allows us to compute $\mathbf{Out}_{tbr}(s)$ using structural information on all assignments which is readily available.

Cascades of Statements. The standard data flow equations apply for cascades of statements $s := [s_1, s_2]$, i.e. $\mathbf{In}_{tbr}(s_1) = \mathbf{In}_{tbr}(s)$, $\mathbf{In}_{tbr}(s_2) = \mathbf{Out}_{tbr}(s_1)$, and $\mathbf{Out}_{tbr}(s) = \mathbf{Out}_{tbr}(s_2)$. $\mathbf{Out}_{tbr}(s_i) = \mathbf{Gen}_{tbr}(s_i) \cup (\mathbf{In}_{tbr}(s_i) \setminus \mathbf{Kill}_{tbr}(s_i))$ for $i = 1, 2$ leads to

$$\mathbf{Out}_{tbr}(s) = \mathbf{Gen}_{tbr}(s_2) \cup (\mathbf{Gen}_{tbr}(s_1) \cup (\mathbf{In}_{tbr}(s) \setminus \mathbf{Kill}_{tbr}(s_1))) \setminus \mathbf{Kill}_{tbr}(s_2)$$

which results in the requirement for explicit expressions for $\mathbf{Gen}_{tbr}(s_i)$ and $\mathbf{Kill}_{tbr}(s_i)$ $(i = 1, 2)$. For example, if both s_1 and s_2 are scalar assignments then (4) and (5) can be used to derive more specific expressions. In fact, equation (8) was derived this way.

Branches Static TBR analysis is conservative, i.e. for a conditional branch statement $s := [\texttt{if}\ (c)\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}]$ we have

$$\begin{aligned} \mathbf{Gen}_{tbr}(s) &= \mathbf{Gen}_{tbr}(s_1) \cup \mathbf{Gen}_{tbr}(s_2) \\ \mathbf{Kill}_{tbr}(s) &= \mathbf{Kill}_{tbr}(s_1) \cap \mathbf{Kill}_{tbr}(s_2) \\ \mathbf{Out}_{tbr}(s) &= \mathbf{Out}_{tbr}(s_1) \cup \mathbf{Out}_{tbr}(s_2) \end{aligned}$$

Loops. Consider a loop $s := [\texttt{while}\ (c)\ \texttt{do}\ s_1\ \texttt{done}]$ where s_1 is a cascade of statements as in Section 4. We are interested in two types of information:

1. Knowing $\mathbf{In}_{tbr}(s)$ we would like to compute $\mathbf{Out}_{tbr}(s)$.
2. For every assignment $s' \in s_1$ we need to determine $\mathbf{Kill}_{tbr}(s')$.

Both 1. and 2. should be obtained with a minimal computational effort. Below we show that for the TBR status information within structured programs at most two traversals of the code inside loops are required. We will specify the index of the traversal as a superscript, i.e. for example $\mathbf{Out}_{tbr}(s_1)^1$ denotes the set of tbr-active variables after a single analysis of s_1. Furthermore, we assume that the control flow is reversed by counting the number of loop iterations performed by the forward code as in the example in Section 1.

Proposition 1 $\mathbf{Out}_{tbr}(s) = \mathbf{Out}_{tbr}(s_1)^1$.

PROOF. Under what circumstances is a variable v in $\mathbf{Out}_{tbr}(s)$?

1. If it is in $\mathbf{In}_{tbr}(s)$ and if it is not overwritten inside s_1, i.e. if $v \in \mathbf{In}_{tbr}(s_1)^1 \wedge \nexists s' \in s_1 : v \in \mathbf{Kill}_{tbr}(s')^1$ or
2. if it becomes tbr-active as a result of executing some assignment $s' \in s_1$ and if it is not overwritten by any assignment $s'' \in s_1$ such that $s'' > s'$, i.e. if $\exists s'' \in s_1 : s'' \in \mathbf{Gen}_{tbr}(s'')^1 \wedge \nexists s' > s'' \in s_1 : v \in \mathbf{Kill}_{tbr}(s')^1$.

We use the notation $s'' > s'$ to indicate that the statement s' is executed before s'' when running the forward code. Obviously, both 1. and 2. can be checked by performing a single analysis of the loop body. ■ A major consequence is that in order to determine $\mathbf{In}_{tbr}(s)$ for some statement within the forward code we have to analyze each statement preceding s only once, even if it is part of a (possibly nested) loop.

In general, for $s' := [v = f(\mathbf{u})]$ we will know whether to record v only after the second traversal of s_1.

Proposition 2 $\forall s' \in s_1 : \mathbf{Kill}_{tbr}(s') = \mathbf{Kill}_{tbr}(s')^2$

PROOF. Consider $s' \in s_1$ where $s' := [v = f(\mathbf{u})]$.

1. If v is tbr-active during the first traversal, i.e. if

$$(v \in \mathbf{In}_{tbr}(s_1) \vee \exists s'' < s' : v \in \mathbf{Gen}_{tbr}(s')) \wedge (\nexists s''' : (s'' < s''' < s' \wedge v \in \mathbf{Kill}_{tbr}(s'''))),$$

   then the decision can be made during the first traversal.
2. If v is tbr-passive
   (a) because there exists an assignment $s'' < s'$ which overwrites v and $\nexists s''' : s'' < s''' \wedge v \in \mathbf{Gen}_{tbr}(s''')$ then the decision can be made during the first traversal;
   (b) because v is not in $\mathbf{In}_{tbr}(s)$ then due to $\mathbf{In}_{tbr}(s_1) = \mathbf{In}_{tbr}(s) \cup \mathbf{Out}_{tbr}(s_1)^1$ we need a second iteration to cover the case $v \in \mathbf{Out}_{tbr}(s_1)^1$.

■

5 Summary, Preliminary Results, Outlook

The generation of adjoint code is based on the reversal of the control flow of the forward code. Adjoint versions of every single assignment contained within the latter have to be generated. The reversed control flow leads to the requirement to access the values of certain intermediate variables in reverse order. In general, due to intermediate variables being overwritten in the forward code, this can only be ensured by storing the corresponding values or recomputing them.

In this paper we have presented some flow equations for propagating the information on whether the value of an intermediate variable has to be recorded,

i.e. whether the current value is required for the generation of a correct adjoint code. Implementations of these ideas showed promising reductions of the memory requirement when following a pure "store all" strategy In [5] the ideas presented in this paper were applied to a large industrial thermal-hydraulic code developed at EDF-DER in France (70000 lines, 500 sub-programs, 1000 parameters). Using the TBR analysis the tape size could be decreased by a factor of 5. The size of the standard tape generated by *Odyssée* version 1.7 [6]: is $213920 \cdot 10^6$ scalar values (or 1711360 MBytes if every value is a double), whereas the optimized tape contains only $40486 \cdot 10^6$ scalar values (or 323888 MBytes).

Our next step will be the generalization of the results presented here to interprocedural TBR analysis and general unstructured programs. In collaboration with colleagues working at INRIA Sophia-Antipolis, France, these ideas are currently being implemented in TAPENADE [2] - the successor of *Odyssée*. A general discussion of optimizing the memory requirements in reverse mode AD is in work.

Acknowledgement

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

1. Fortran standard. *International Organization for Standardization*, 1 (1997), 2 (2000), 3 (1999)(ISO/IEC 1539).
2. `http://www-sop.inria.fr/tropics/`. URL.
3. A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.
4. G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms - From Simulation to Optimization*, to appear in Springer LNCS, New York, 2001.
5. C. Faure and U. Naumann. The taping problem in Automatic Differentiation. In *[4]*.
6. C. Faure and Y. Papegay. *Odyssée* user's guide. version 1.7. Technical Report 0224, INRIA, September 1998.
7. R. Giering and T. Kaminski. Towards an optimal trade-off between recalculation and taping in reverse mode ad. In *[4]*. to appear in Springer LNCS, 2001.
8. A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse Automatic Differentiation. *Optimization Methods and Software*, (1):35–54, 1992.
9. A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation.* SIAM, Philadelphia, April 2000.
10. R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation.* ACM, 2000.